Introdução a Complexidade de Algoritmos

Profa. Juliana Kaizer Vizzotto

Complexidade de Algoritmos - Aula 2

Roteiro

- Revisão da aula passada
- Custo de Algoritmos
- Análise Assintótica

Conceitos Básicos

Análise do *Insertion-sort*

```
Insertion-sort(A)                          custo  vezes
for j<- 2 to comprimento[A]                c1     n
  do chave <- A[j]                         c2     n-1
  i <- j - 1                               c4     n-1
  while i > 0 e A[i] > chave               c5     S (j=2 n) tj
    do A[i+1] <- A[i]                      c6     S (j=2 n) (tj -1)
       i <- i-1                            c7     S (j=2 n) (tj -1)
  A[i+1] <- chave                          c8     n-1
```

Conceitos Básicos

Análise do *Insertion-sort*

- O tempo de execução do algoritmo é a soma dos tempos de execução para cada instrução realizada
- Tempo de execução do *insertion-sort*

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

- Mesmo para entradas de um dado tamanho, o tempo de execução de um algoritmo pode depender de **qual** entrada desse tamanho é dada.
- Qual seria o **melhor caso** para o *insertion-sort* para uma entrada de tamanho n?

Conceitos Básicos

Análise do *Insertion-sort*

- No melhor caso $t_j = 1$, logo o tempo de execução será:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

- Esse tempo de execução pode ser expresso como $an + b$ para constantes a e b que dependem dos custos de instrução
- Assim, o cálculo nos dá uma **função linear** em n

Análise do *Insertion-sort*

- Qual seria o **pior caso**? Temos que $t_j = j$.
- Série aritmética:

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2}.$$

- Como $\sum_{j=2}^{n} j = (\sum_{j=1}^{n} j) - 1 = (\frac{n(n+1)}{2}) - 1$.
- Considerando $k = j - 1$, temos

$$\sum_{j=2}^{n} (j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}.$$

Conceitos Básicos

Análise do *Insertion-sort*

- Logo, no pior caso:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n-1)}{2} - 1)$$
$$c_6(\frac{n(n-1)}{2}) + c_7(\frac{n(n-1)}{2}) + c_8(n-1)$$

- Tempo de execução no pior caso pode ser expresso como $an^2 + bn + c$, i.e., função quadrática (ou tb polinomial).

Solução exercício: busca linear

```
for i=1 to A.lenght
    if A[i]==v
      return i
return NIL
```

- Propriedade do Loop invariante: *No início de cada iteração do loop for, o subarray* $A[1..i-1]$ *consiste de todos os elementos que não são iguais a* v.

Solução exercício: busca linear

- **Inicialização:** inicialmente o subarray está vazio. Assim, nenhum dos seus elementos é igual a v.
- **Manutenção:** Na iésima iteração, verifica-se quando $A[i]$ é igual a v. Se sim, o loop termina e caso contrário, a iteração continua. Logi, se o subarray $A[1..i-1]$ não contém v antes da iésima iteração, o subarray $A[1..i]$ não conterá antes da próxima iteração. A menos que a iésima iteração termine o loop.
- **Término:** o loop termina em um dos seguintes casos:
  - o algoritmo encontrou o índice i tal que $v = A[i]$.
  - o algoritmo chegou ao final do array, i.e., não encontrou v no array A. Portanto, retornou NIL.

Custo de Algoritmos

- O nosso foco em termos de custo será: tempo de execução
- Um definição razoável é considerar algoritmos em tempo polinomial como eficientes (tese de Cobham-Edmonds)
- A complexidade pode ser vista como uma propriedade do problema (algoritmo)
- A análise da complexidade de um algoritmo consiste em determinar o número de operações básicas (atribuição, soma, comparação) em relação ao tamanho da entrada

Conceitos Básicos

Análise do pior caso e do caso médio

- Normalmente, vamos nos concentrar apenas na descoberta do tempo de execução do **pior caso**, i.e., o tempo de execução mais longo para qualquer entrada n.
  - Conhecer o tempo de execução no pior caso nos dá garantia de que o algoritmo nunca irá demorar mais tempo.
  - Para alguns algoritmos o pior caso ocorre com bastante frequência. Por exemplo, na pesquisa de um banco de dados em busca de um determinado fragmento de informaçã, o pior caso de pesquisa ocorrerá frequentemente quando a informação não estiver presente no banco de dados.
  - Muitas vezes o caso médio é tão ruim quanto o pior caso.

Análise Assintótica

- Em geral o número de operações fornece um nível grande de detalhamento
- Portanto, analisamos somente a taxa ou ordem de crescimento, subs- tituindo funções exatas com cotas mais simples.
- Exemplo:
  - Um algoritmo com número de operações: $an^2 + bn + c$
  - Para análise assintótica não interessam: os termos de baixa ordem e os coeficientes constantes
  - Logo o tempo de execução tem cota n^2, denotado com $O(n^2)$

Análise Assintótica

- Considere dois algoritmos A e B com tempo de execução $O(n^2)$ e $O(n^3)$, respectivamente. Qual deles é o mais eficiente?
- Esperamos que o primeiro seja mais eficiente que o segundo. Para n grande, isso é verdadeiro, mas o tempo de execução atual pode ser $100n^2$ no primeiro e $5n^3$ no segundo. Logo, para $n < 20$ o segundo algoritmos é mais rápido.
- **Assintoticamente** consideramos um algoritmo com complexidade $O(n^2)$ melhor que um algoritmo com $O(n^3)$.
- De fato, para n suficientemente grande $O(n^2)$ sempre é melhor.

Análise Assintótica

- Considere dois algoritmos A e B com tempo de execução $O(n^2)$ e $O(n^3)$, respectivamente. Qual deles é o mais eficiente?
- Esperamos que o primeiro seja mais eficiente que o segundo. Para n grande, isso é verdadeiro, mas o tempo de execução atual pode ser $100n^2$ no primeiro e $5n^3$ no segundo. Logo, para $n < 20$ o segundo algoritmos é mais rápido.
- **Assintoticamente** consideramos um algoritmo com complexidade $O(n^2)$ melhor que um algoritmo com $O(n^3)$.
- De fato, para n suficientemente grande $O(n^2)$ sempre é melhor.

Análise Assintótica

- Exemplo: **insertion sort**, leva $c_1 n^2$ para ordenar n itens. **Merge sort** leva $c_2 n lgn$.
- Insertion sort, tipicamente, tem um fator constante menor que o merge sort, i.e., $c_1 < c_2$.
- Considere, insertion sort $c_1 n * n$ e merge sort $c_2 n * lgn$.
- Quando $n = 1000$, lgn é aproximadamente 10 e quando n é igual a um milhão, lgn é aproximandamente 20.
- Embora o insertion sort seja mais rápido para entradas pequenas, quando o tamanho de n fica grande, a vantagem do lgn versus n compensa muito.

Análise Assintótica

- Considere um computador rápido, A, executando *insertion sort* e um computador mais lento, B, executando *merge sort*.
- Cada um deve ordenar um array com 10 milhões de números.
- Suponha que o computador A execute 10 bilhões de instruções por segundo!!!!
- E o computador B execute somente 10 milhões de instruções por segundo.
- O computador A é 1000 vezes mais rápido que B.

Análise Assintótica

- Um ótimo programador fez um insertion sort em código de máquina para o computador A, que requer somente $2n^2$ instruções para ordenar n números.
- Um programador mediano implementou merge sort usando uma linguagem de alto nível com tempo de $50nlgn$ intruções.
- Assim, para ordenar 10 milhões de números o computador A leva:

$$\frac{2 * (10^7)^2 \ instructions}{10^10 \ instructions/sec} = 20.000 \ secs(\text{mais de 5.5 horas})$$

- Enquanto que o computador B leva:

$$\frac{50 * 10^7 lg10^7 \ instructions}{10^7 \ instructions/sec} = 1163 secs(\text{menos de 20 min})$$

Análise Assintótica

- Usando um algoritmo, o qual o tempo de execução cresce mais lentamente, mesmo o computador B, executa 17 vezes mais rápido que um computador super rápido como A.
- A vantagem do merge sort fica muito mais notória quando ordenamos um array com 100 milhões de números. Insertion sort poderia levar mais que 20 dias, merge sort leva menos de 4 horas.

Análise Assintótica

Um panorama do tempo de execução

- Tempo constante: $O(1)$: raro!
- Tempo sublinear: $\log(n), \log(log(n))$: Rápidos!!
- Tempo linear: número de operações proporcional à entrada
- Tempo $n \log n$: comum em algoritmos de divisão e conquista
- Tempo polinomial n^k: frequentemente de baixa ordem ($k \leq 10$): considerado eficiente
- Tempo exponencial: $2^n, n!, n^n$ considerado intratável

Análise Assintótica

Um panorama do tempo de execução

- Tempo constante: $O(1)$: raro!
- Tempo sublinear: $\log(n), \log(log(n))$: Rápidos!!
- Tempo linear: número de operações proporcional à entrada
- Tempo $n \log n$: comum em algoritmos de divisão e conquista
- Tempo polinomial n^k: frequentemente de baixa ordem ($k \leq 10$): considerado eficiente
- Tempo exponencial: $2^n, n!, n^n$ considerado intratável

Análise Assintótica

Exemplos:

- Tempo constante: determinar se uma sequência de números começa com 1.
- Tempo sublinear: busca binária
- Tempo linear: buscar o máximo em uma sequência
- Tempo $n \log n$: *mergesort*
- Tempo polinomial: multiplicação de matrizes
- Tempo exponencial: busca exaustiva de todos os subconjuntos de um conjunto, de todas as permutações de uma sequência

Problemas super polinomiais?

- Consideramos a classe P de problemas com solução em tempo polinomial tratável
- NP é outra classe importante qu contém muitos problemas práticos (e a classe P)
- Não se sabe se todos possuem algoritmo eficiente
- Problemas $NP - completos$ são mais complexos que os NP: se um deles tem uma solução eficiente toda a classe tem!
- Vários problemas $NP - completos$ são parecidos com problemas que têm algoritmos eficientes.
  - Ciclo euleriano, ciclo hamiltoniano
  - Caminhos mais curtos, caminhos mais longo
  - Satisfatibilidade 2-CNF,Satisfatibilidade 3-CNF

Exercícios

- Considere a ordenação de n números armazenados no arranjo A, localizando primeiro o menor elemento de A e permutando esse elemento com o elemento contido em $A[1]$. Em seguida, encontre o segundo menor elemento de A e troque pelo elemento A[2]. Continue dessa maneira para os primeiros $n-1$ elementos de A. Escreva o pseudocódigo para esse algoritmo, conhecido como *ordenação por seleção*. Por que ele só precisa ser executado para os primeiros n-1 elementos? Forneça os tempos de execução no melhor caso e no pior caso da ordenação por seleção.

Projeto de Algoritmos

- Existem muitas maneiras de projetar algoritmos
- A ordenação por inserção utiliza uma abordagem **incremental**: tendo ordenado o subarray $A[1..j-1]$, inserimos o elemento isolado $A[j]$ em seu lugar apropriado
- Vamos examinar uma abordagem de projeto alternativa, conhecida como **dividir e conquistar**
- Vamos utilizar essa abordagem para projetar um algoritmo de ordenação cujo tempo de execução do pior caso é muito menor que o da ordenação por inserção. item Uma vantagem dos algoritmos de dividir e conquistar é que seus tempos de execução são frequentemente fáceis de determinar como veremos durante o semestre

Projeto de Algoritmos

Dividir e conquistar

- Lembrando...*recursão*...pense fatorial!! fatorial recursivo!!
- Idéia: desmembrar o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam as soluções para criar uma solução para o problema original.
- O problema de dividir e conquistar envolve três passos em cada nível de recursão:
  - **Dividir** o problema em um determinado número de subproblemas
  - **Conquistar** os subproblemas, resolvendo-os recursivamente.
  - **Combinar** as soluções dadas aos subproblema, a fim de formar a solução para o problema original.

Dividir e conquistar: ordenação por intercalação (*mersort*)

- **Dividir** divide a sequência de n elementos a serem ordenados em duas subsequências de $n/2$ elementos cada uma
- **Conquistar** classifica as duas subsequências recursivamente, utilizando a ordenação por intercalação
- **Combinar** faz a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada

A operação chave do algoritmo é a intercalação de duas sequências ordenadas. Utilizamos um procedimento auxiliar Merge(A,p,q,r), tal que $p \leq q \leq r$

Dividir e conquistar: mergesort

```
Merge(A,p,q,r)
n1 <- q - p + 1
n2 <- r - q
criar arrays L[1..n1+1] e R[1 ...n2+1]
for i <- 1 to n1
   do L[i] <- A[p+ i-1]
for j <- 1 to n2
   do R[j] <- A[q+j]
L[n1+1]<- infinito
R[n2+1] <- infinito
i <- 1
j <- 1
...
```

Dividir e conquistar: mergesort

```
...
for k <- p to r
  do if L[i] <= R[j]
        then A[k] <- L[i]
             i <- i + 1
        else A[k] <- R[j]
             j <- j +1
```

Dividir e conquistar: mergesort

Exemplo

- Execute o pseudocódigo acima na chamada Merge(A,9,12,16) quando o subarray A[9..16] contém a sequência $[2, 4, 5, 7, 1, 2, 3, 6]$